Forensic Analysis of the Windows Registry in Memory

By

Brendan Dolan-Gavitt

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2008 USA

Baltimore, MD (Aug 11th - 13th)

available at www.sciencedirect.com

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

Forensic analysis of the Windows registry in memory[☆]

Brendan Dolan-Gavitt

MITRE Corporation, 202 Burlington Road, Bedford, MA, USA

Keywords:
Digital forensics
Microsoft Windows
Volatile memory
Registry
Cached data

ABSTRACT

This paper describes the structure of the Windows registry as it is stored in physical memory. We present tools and techniques that can be used to extract this data directly from memory dumps. We also provide guidelines to aid investigators and experimentally demonstrate the value of our techniques. Finally, we describe a compelling attack that modifies the cached version of the registry without altering the on-disk version. While this attack would be undetectable with conventional on-disk registry analysis techniques, we demonstrate that such malicious modifications are easily detectable by examining memory.

1. Introduction

The Windows registry is a hierarchical database used in the Windows family of operating systems to store information that is necessary to configure the system (Microsoft Corporation, 2008). It is used by Windows applications and the OS itself to store all sorts of information, from simple configuration data to sensitive data such as account passwords and encryption keys. As described in Section 2, researchers have found that the registry can also be an important source of forensic evidence when examining Windows systems.

Another important yet non-traditional source of forensic data is the contents of volatile memory. By examining the contents of RAM, an investigator can determine a great deal about the state of the machine when the image was collected. Although techniques for analyzing and extracting meaningful information from the raw data found in memory are still relatively new, guidance on the collection of physical memory is now a common part of many forensic best practice documents, such as the NIST Special Publication ''Guide to Integrating Forensic Techniques into Incident Response'' (Kent et al., 2006).

Our work seeks to bring these two areas of research together by allowing investigators to apply registry analysis techniques to physical memory dumps. We will begin by explaining the structure of the Windows registry as it is represented in memory, and describe techniques for accessing the registry data stored in memory. A prototype implementation of an in-memory registry parser will then be presented, along with some experimental results from several memory images. We will also discuss particular considerations investigators should be aware of when looking at the registry in memory.

Finally, we will show that although under normal conditions the stable keys (see Section 3.3 for details on the distinction between stable and volatile keys) recovered from the in-memory copy of the registry are essentially a subset of those found in the on-disk copy, an attacker with access to kernel memory can alter the cached keys and leave those on disk unchanged. The operating system will then make use of the cached data from the registry, and a forensic examination of the disk will not detect the changes. We will show how analyzing the registry in memory can detect this attack.

2. Related work

Over the past several years, it has become increasingly clear that the registry can contain a great deal of information that is of use to forensic examiners. Research has shown that it contains such data as lists of recently run programs (Stevens,

[☆] The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the author.

1742-2876/$ – see front matter © 2008 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2008.05.003

2006), logs of devices recently attached to the system (such as USB keys) (Carvey, 2007), and wireless networks the user connected to (Carvey, 2005a). Registry data, Carvey notes, "provides a wealth of information that the investigator can use to make his case" (Carvey, 2005a).

Meanwhile, starting with the DFRWS 2005 Memory Analysis Challenge (DFRWS, 2005), a great deal of progress has been made towards cataloging the contents of physical memory on Windows systems and documenting how to make use of the information it contains. Using freely available tools, investigators can find disk encryption keys (Walters and Petroni, 2007), list processes and threads (Schuster, 2006), and detect the presence of some techniques used by malicious software such as DLL injection and hiding (Dolan-Gavitt, 2007; Walters, 2006).

Attacks that alter cached data in memory to deceive the operating system while remaining hidden from tools that analyze non-volatile storage are not entirely unheard of. In their 2006 USENIX paper, Petroni et al. (2006) described an attack which modifies SELinux's access vector cache to grant additional privileges while remaining undetected by traditional kernel integrity protections. More broadly, attacks on cached data fall into the category of in-memory operating system alterations; other examples of this type of attack include Metasploit's DLL injection payload (Metasploit, 2008), which loads executable code into the address space of an exploited process, and the Slammer worm, which was never written to disk as it propagated from system to system (F-Secure, 2003).

Documentation about the inner workings of the Configuration Manager, the Windows subsystem that manages the registry, is sparse. The most complete reference is *Windows Internals*, by Russinovich and Solomon (2004), which describes some of the internal mechanisms, but does not provide enough detail, on its own, to allow extraction of the registry from memory. A blog post by Anand (2008) provides some more details, and gives an example of manually translating a cell index into a virtual address using WinDbg. Finally, a series of posts by Dolan-Gavitt (2008a–c) provide lower-level technical details on the internal mechanisms of the Configuration Manager.

3. The registry in memory

3.1. *Overview*

Although the Windows registry appears as a single hierarchy in tools such as regedit, it is actually made up of a number of different binary files called *hives* on disk. These files and their relationship to the hierarchy normally seen are described in KB256986 (Microsoft Corporation, 2008). The hive files themselves are broken into fixed sized *bins* of `0 × 1000` bytes, and each bin contains variable-length *cells*, which hold the actual registry data. References in hive files are made by *cell index*, which is essentially a value that can be used to derive the location of the cell containing the referenced data.

As for the structure of the registry data itself, it is generally composed of two distinct data types: *key nodes* and *value data*. The structure can be thought of as analogous to a filesystem, where the key nodes play the role of directories and the values act as files.[1] One key difference, however, is that data in the registry always has an explicit associated type, whereas data on a filesystem is generally only weakly typed (for example, through a convention such as file extension).

To work with registry data in memory, it is necessary to find out where in memory the hives have been loaded and know how to translate cell indexes to memory addresses. It will also be helpful to understand how the Windows Configuration Manager works with the registry internally, and how we can make use of its data structures to tell us what the operating system itself maintains about the state of the registry.

3.2. *Locating hives*

The Configuration Manager in Windows XP references each hive loaded in memory using the `_CMHIVE` data structure.[2] The `_CMHIVE` contains several pieces of metadata about the hive, such as its full path, the number of handles to it that are open, and pointers to the other loaded hives on the system (using the standard `_LIST_ENTRY` data structure used in many Windows kernel structures to form linked lists). It also has another important structure embedded within it, the `_HHIVE`, which contains the mapping table used to translate cell indexes (more details on this are given in Section 3.3).

Our approach to finding hives in memory has two stages. First, we scan physical memory to find a single hive; this is easily accomplished, as each `_HHIVE` begins with a constant signature `0 × bee0bee0` (a little-endian integer). Furthermore, the structure is allocated from the kernel's paged pool, and has the pool tag `CM10`; these two indicators are sufficient to find valid `_HHIVE`s in all Windows XP images we have examined. Once a single instance has been found, the `HiveList` member is used to locate the others in memory.[3] The pointers to the previous and next hives in the list are virtual addresses in kernel memory space, and must be translated to physical addresses using the page directory of some process.[4]

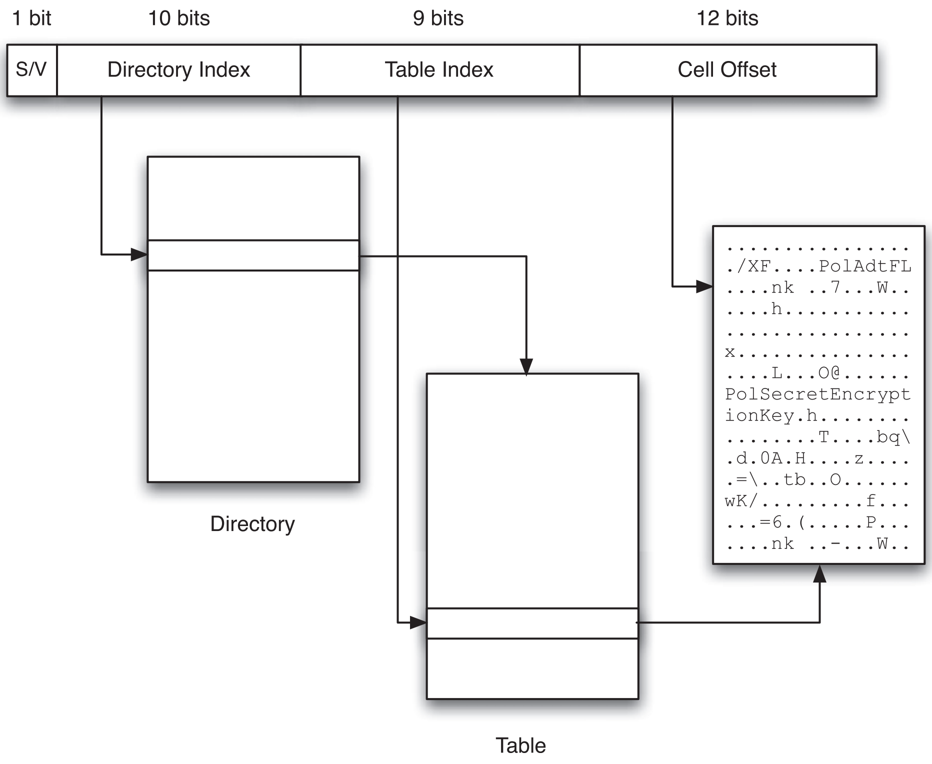
In typical Windows XP SP2 memory images, we found 13 hives: the NTUSER and UsrClass hives for the currently logged on user, the LocalService user, and the NetworkService user (total of six hives); the template user hive ("default"); the Security Accounts Manager hive ("SAM"); the system hive; the SECURITY hive; the software hive; and, finally, two volatile hives that have no on-disk representation. The two volatile hives deserve some special mention: one, the `HARDWARE` hive, is generated at boot and provides information on the hardware detected in the system. The other, the `REGISTRY` hive, contains only two keys, `MACHINE` and `USER`, which are used to provide a unified namespace in which to attach all other hives.

[1] This analogy is borrowed from Farmer (2007), though it probably does not originate with him.

[2] Data structures referenced in this paper can be found in the public debug symbols for Windows XP Service Pack 2 unless otherwise noted, and are viewable with the `dt` command in WinDbg.

[3] This is substantially the same process described by Dolan-Gavitt (2008b).

[4] Since the kernel-space portion of virtual address space is the same for all processes, any process will do.



```
................
./XF....PolAdtFL
....nk ..7...W..
....h...........
................
x...............
....L...O@......
PolSecretEncrypt
ionKey.h........
........T....bq\
.d.0A.H....z....
.=\..tb..O......
wK/.........f...
...=6.(.....P...
....nk ..-...W..
```

Fig. 1 – Using a cell index with the hive's address tables to derive a virtual address.

3.3. *Cell indexes*

Unlike their layout on disk, hive files in memory need not be contiguous. Moreover, keys and values may be added while the operating system is running, and while it would be inefficient to have to search for free space in the registry file, simply appending a new bin to the end of the hive would quickly cause the hive to grow to an unmanageable size. To solve this, the Configuration Manager creates a mapping between cell indexes and addresses in virtual memory, in much the same way that a process gets a map between its virtual addresses and the physical address space of main memory (Russinovich and Solomon, 2004).

The map for a given hive is stored in the `_HHIVE` structure's `Storage` member, and once located, it can be used to give us full access to the registry data stored in memory. To perform the translation, the cell index is broken into four parts: a one bit selector saying whether the cell index's main storage is stable (on-disk) or volatile (only in memory), 10 bits giving the index into the list of hive address tables, 9 bits that select a single entry from that table, and finally a 12-bit offset into the block given by that table entry. Fig. 1 shows how the different parts of the cell index are related to the hive's address tables and cell data. Now, given the address of the table directory and a cell index, we can translate that cell index into a virtual address in kernel memory (Dolan-Gavitt, 2008a).

Once we can perform this translation, reading the registry is relatively straightforward, as the data structures are identical to those used for the on disk. Several open source programs exist that can read binary registry data, such as Samba's regfio (Samba) or the perl module Parse::Win32Registry (Macfarlane), and the data structures and algorithms from these tools can be applied directly to the task of reading registry data in memory with only small modification to use the in-memory translation method for cell indexes. To walk the entire hive, we can start at the root key (always at cell index 0×20), and then walk the subkeys just as we would an on-disk hive.

One crucial difference, however, is the existence of volatile keys and values in in-memory hives. The `_CM_KEY_NODE` structure has two members, `SubKeyCounts` and `SubKeyLists`, that give the number of subkeys and a pointer to the subkey list, respectively. Each member, however, is actually an array of length two: the first entry in the array refers to the *stable* keys, while the second refers to the *volatile* keys. Most existing implementations of Windows registry parsers, such as Samba's regfio library (Samba) and the Perl Parse::Win32Registry module (Macfarlane), do not handle volatile subkey lists, and describe those portions of the key structure as "unknown." The registry implementation in ReactOS handles volatile keys correctly, however.

These volatile keys are never stored on disk, and are automatically generated by the operating system when the machine is booted. Examples of information stored in these keys include an enumeration of all hardware detected on the system, the volatile portions of a user's environment, mounted volumes, and the current machine name. Although it is possible to access this information while the system is booted using live response techniques, it cannot be recovered using the on-disk hives from an image of the system.[5] Using these techniques, however, an investigator will be able to

[5] Carvey (2005b) notes this problem in a blog post when describing how to find out what user mounted a particular volume; this information is only available in a volatile registry key.

access the volatile keys and values in a stable, repeatable way by examining a dump of physical memory.

3.4. *Management mechanisms*

In addition to simply extracting as many keys and values from the cached copy of the registry in memory as possible, an investigator might wish to gain an understanding of what data the Configuration Manager was working with. For example, the examiner might wish to know what keys were open on the system, and how many things were referencing them. To answer these questions, one can make use of several data structures used by the Configuration Manager to provide fast access to currently open keys.

When an application attempts to open a key, the Configuration Manager must be able to quickly determine if the key is already open, and if so, return a handle to the same object, to ensure that all applications are always referencing the same data. To accomplish this, each open key, as well as all of its ancestors, has associated with it a *Key Control Block* (data structure `_CM_KEY_CONTROL_BLOCK`), or KCB, that keeps track of its reference count, last write time, and cell index. The different handles that each process gets will then all point to the same KCB.

To satisfy the requirement that finding a KCB for a given key be fast, the Configuration Manager uses a hash table to keep track of all the KCBs. The address of this table is given by the kernel global variable `CmpCacheTable` and its size by `CmpHashTableSize`; the address of these variables can be found either by using the debug symbols for the kernel that was loaded in the memory image, or by searching through the mapped kernel image and using some heuristic to validate whether a given address is a pointer to the hash table.[6] Each entry is a pointer to the `KeyHash` member of a Key Control Block, which is of type `_CM_KEY_HASH`. Entries in the hash table can point to more than one Key Control Block, and the full list for a given table entry can be found by repeatedly following the `NextHash` pointer in the key hash structure (Dolan-Gavitt, 2008c).

4. Implementation

We have implemented the techniques described in Section 3 as a plugin for Volatility (Walters, 2007). This allows us to make use of the pre-existing libraries to do virtual address translation, process listing, and so on, as well as easily define the data structures needed to parse registry data (through the data model exposed in `vtypes.py`).

Each hive is represented as its own *virtual address space*; the cell indexes described earlier are treated as memory addresses within the space of the hive. Thus when we want to find the root of a hive, it suffices to get the `_CM_KEY_NODE` structure at hive address 0×20; the hive address space object then handles translating that cell index into a virtual address, and the virtual address space object in turn translates the virtual address into a physical offset in the memory dump.

A small library of functions has been developed that handles the most common tasks associated with reading information from registry hives: reading keys and values, opening a key given its full path, getting the root of the hive, and so on. Using these functions, other researchers should be able to easily create their own plugins to extract and Interpret portions of the registry that have forensic relevance, such as the UserAssist keys (Stevens, 2006). Plugins have also been developed to automate the process of finding hives and examining the Key Control Blocks used by the Configuration Manager.

5. Experimental results

Several experiments were performed to compare the effectiveness of various methods of examining registry data. We tested four images running Windows XP Service Pack 2:

1. The first image from the Computer Forensic Reference Data Sets (CFReDS) project (NIST) (512 M of RAM), filename xp-laptop-2005-06-25.img.
2. The second CFReDS image (512 M of RAM), filename xp-laptop-2005-07-04-1430.img.
3. A recently booted VMWare image with 512 M of RAM.
4. A standard desktop machine with 1 GB of RAM that had been up for 8 days, and had been in use for 541 days.

For each image, we counted the number of open keys and values in the following ways:

- Number of Key Control Blocks in the cache table as described in Section 3.4.
- Number of unique keys in process handle tables.
- Number of values found by searching pool allocations (using Schuster's (2007) PoolTools).
- Number of keys found by walking each hive from the root key.
- Number of values found by walking each hive from the root key.
- When available, number of keys and values in the on-disk hives.

The results are summarized in Table 1. The data reveal a few interesting facts: first, our technique of walking the hive starting at the root in memory is more effective than any other existing method of extracting registry keys and values from RAM. Second, on a real workstation, the amount of registry data recoverable from memory may be substantially smaller than what was observed in a virtual machine image; this is likely because of the large number of applications running, which would have caused unused portions of the registry to be paged out to disk (see Table 2 for a comparison of how much registry data was unreadable in each image). Finally, we note that an average of 631 keys and 1231 values per image were volatile and would not have been found using methods that only examine the hives on disk.

We also performed an experiment to determine how much difference there was between the on-disk version of the

[6] One such heuristic is given by Dolan-Gavitt (2008c), and involves checking to see if the resulting KCBs in the candidate table have a `KeyHive` member that points to a hive with the correct signature.

Table 1 – Number of keys/values recoverable using various techniques

	NIST XP 6/25	NIST XP 7/4	VMWare image	Standard desktop
KCBs	3367	3998	2981	2703
Keys (memory,stable)	81,996	91,072	71,533	26,431
Keys (memory,volatile)	580	581	485	879
Values (memory,stable)	127,578	151,664	121,574	84,781
Values (memory,volatile)	1162	1165	963	1633
Keys (handles)	278	260	194	355
Values (pool)	7530	7424	4945	14,362
Keys (disk)	N/A	N/A	80,298	166,543
Values (disk)	N/A	N/A	144,468	315,054
Processes	47	45	23	128

registry and the in-memory copy. We copied the registry hives from a suspended VMWare image, along with its memory image, and then enumerated all keys in each hive, both in-memory and on disk, and compared the outputs. Across all hives, we found:

- 546 keys were found only in memory. These are the volatile keys described in Section 3.3.
- 8246 keys were missing from the memory view, either because the Configuration Manager had not mapped that portion of registry hive in, or because it had been mapped into memory and then paged out.
- Six keys had a more recent timestamp in memory than on disk, indicating that a write had occurred but the change had not yet been written to disk.

6. Guidelines for investigators

In general, any standard forensic methods for examining registry data can be applied to the registry data in memory. However, there are certain caveats that apply uniquely to the examination of the data cached in RAM. Most importantly, it cannot be assumed that the data found in memory is complete. In addition to the usual consideration that the contents of main memory may be swapped out to the page file, it is also possible that parts of the registry may have never been brought into memory in the first place. In this case, its entry in the cell map table (see Section 3.3) will be set to zero, and the data is unlikely to be found in memory.

The most immediate consequence of this is that tools that deal with the registry from memory must be robust and able to handle missing data without crashing. Our initial attempts to work with registry data stored in memory involved extracting the hive from memory and saving it to disk, writing NULLs when data from memory was missing. However, we found that existing tools were unable to deal with the missing sections, and would crash or display incomplete output, and we were instead forced to write our own parser that was able to detect invalid structures and ignore them.

Because not all of the registry will necessarily be in memory, the data recovered from RAM should be examined alongside the on-disk hives. This will allow the investigator to get a complete picture of the registry: every key and value will be available on disk, and data that has not yet been written to disk can also be recovered from memory. In addition, as we will discuss in the next section, it is possible for an attacker to alter the data stored in the hive in memory without modifying the copy on disk; however, we will demonstrate that comparing the two views will reveal this activity.

7. Detecting the cached data attack

As seen in Section 5, the so-called "stable" registry data in memory is generally a subset of what can be found on disk, due to the fact that any modified keys or values are written out to disk every five seconds by default (Russinovich and Solomon, 2004). However, we have found that it is possible for an attacker with the ability to modify kernel memory to alter the cached registry data in memory, and thus alter the behavior of the operating system, without the changes being visible in the on-disk storage. For example, an attacker could find the key in memory that holds the password hashes for the Administrator user, and replace them with precomputed hashes for a known password. The attacker would then be able to log in as Administrator using the password of his choice.

We have tested this attack using WinDbg on a VMWare virtual machine (though it could be accomplished fairly easily by any piece of code with access to kernel memory). First, we located the virtual memory address of the key `SAM\Domains\Account\Users\000001f0`, and determined the address of its "V" value (this is the data value that contains the password hashes for the Administrator account). Then, we used the `eb` command to overwrite the hashes with our own precomputed LanMan and NT hashes for the password "foobar".[7] After logging out of the account to allow the new value to be read by the logon process, we were able to log in with the password "foobar" through both the standard Windows login screen

[7] The process of calculating the hashes is fairly complex and not germane here; for a good discussion see Dolan-Gavitt (2008d).

Table 2 – Number of 0 × 1000 byte blocks readable from hives in memory

Image	Blocks unreadable	Total blocks	Percent unreadable
NIST 6/25	37	6515	0.56
NIST 7/4	2	6542	0.03
VMWare image	103	5838	1.76
Standard desktop	6663	13,159	50.63

and Remote Desktop. We then used system normally for the next 15 minutes (ample time for the Configuration Manager's hive flush mechanism to take effect), created a memory dump by pausing the VM and copying its `vmem` file, and rebooted the virtual machine to verify that the new hashes had not been saved.

As expected, the hashes had not been written out to the disk hive, and the original password was once again in effect. The changes were not flushed to the hive on disk because the modification was made without using the Configuration Manager's normal mechanisms, which update a list of "dirty" bins when registry data is written and schedule a hive flush (Russinovich and Solomon, 2004). To verify that it was possible to detect this modification, we then extracted the cached value of the "V" key from the saved memory dump, and compared it with the version on disk, and thus detected the inconsistency. An examiner looking only at the on-disk hive would have found nothing amiss in this situation.

8. Future work

At the moment, the tools we have developed for accessing the registry in memory only work with images from Windows XP SP2. In order to be useful to the widest range of users, support for other versions of Windows such as Windows 2000 and Vista should be added. Luckily, it appears that the basic mechanisms such as cell maps are essentially the same from Windows 2000 onward; however, the process for finding the `_CMHIVE` structures is likely to be somewhat different. Also, the address of `CmpCacheTable` will not be the same as on Windows XP SP2, and it will have to be derived from the debug symbols on these new platforms.

In addition, the prototype implementation we have currently still involves many manual steps to compare hives. Ideally, this process should be automated, and produce a report showing exactly what data differs between the hive in memory and its counterpart on disk. This would also make detection of any attacks on cached data trivial.

Finally, one limitation of the current techniques is that in order to read a key, all of the key's ancestors in the hive must be accessible. This is because at the moment, our only means of navigating to a key is to start at the root and work our way downward. Thus if a single key is unreadable, the entire subtree below that point will not be read. To solve this, we might imagine instead doing a linear scan across they hive for keys, subkey lists, and values, reassembling as many subtrees as possible. These subtrees could even be reattached to the main tree structure, provided there is only one missing key in between: if key A lists a subkey at cell index B, and key C lists its parent at index B, the presence of B can be inferred even the key itself is unreadable.

9. Conclusion

With the techniques described in this paper, full access to the registry data cached in memory is now possible. We have demonstrated that the format of the registry data in memory is identical to the structure found on disk, but with a different address translation mechanism. Because of this, investigators will be able to leverage all of the traditional forensic methods that apply to the on-disk registry, with some caveats (see Section 6).

Finally, we have shown that there are attacks that cannot be detected without examining the registry in memory, and explored one specific case where an attacker could modify account credentials in the registry without leaving traces on disk. To counter such attacks, we recommended collecting registry data from both RAM and the hard drive, and comparing the two to obtain a complete picture of the state of the Windows registry. Although we are not aware of any incidents involving the attacks described in this paper in the wild, examiners will now be forearmed with tools that can counter them.

By supplementing traditional registry and memory analysis with these techniques, we hope that investigators will gain new tools with which to understand incidents.

REFERENCES

ReactOS, <http://www.reactos.org/en/index.html>.

Anand G. Internal structures of the Windows registry. <http://blogs.technet.com/ganand/archive/2008/01/05/internal-structures-of-the-windows-registry.aspx>, 2008.

Carvey H. The Windows registry as a forensic resource. Digital Investigation 2005a;2(3):201–5.

Carvey H. Registry mining. <http://windowsir.blogspot.com/2005/01/registry-mining.html>, 2005b.

Carvey H. Windows forensic analysis. Norwell, MA, US: Syngress, ISBN 159749156X; 2007.

DFRWS. The DFRWS 2005 forensic challenge. <http://www.dfrws.org/2005/challenge/index.html>, 2005.

Dolan-Gavitt B. The VAD tree: a process-eye view of physical memory. Digital Investigation, http://dfrws.org/2007/proceedings/p62-dolan-gavitt.pdf, September 2007;4:62–4.

Dolan-Gavitt B. Cell index translation.<http://moyix.blogspot.com/2008/02/cell-index-translation.html>, 2008a.

Dolan-Gavitt B. Enumerating registry hives. <http://moyix.blogspot.com/2008/02/enumerating-registry-hives.html>, 2008b.

Dolan-Gavitt B. Reading open keys. <http://moyix.blogspot.com/2008/02/reading-open-keys.html>, 2008c.

Dolan-Gavitt B. SysKey and the SAM. <http://moyix.blogspot.com/2008/02/syskey-and-sam.html>, 2008d.

F-Secure. F-Secure virus descriptions: slammer. <http://www.f-secure.com/v-descs/mssqlm.shtml>, 2003.

Farmer DJ. A forensic analysis of the Windows registry. <http://eptuners.com/forensics/Index.htm>, 2007.

Kent K, Chevalier S, Grance T, Dang H. NIST special publication 800-86: guide to integrating forensic techniques into incident response. 2006.

Macfarlane J. Parse:Win32Registry. <http://search.cpan.org/jmacfarla/Parse-Win32Registry-0.30/>.

Metasploit. Metasploit framework user guide. <http://www.metasploit.com/documents/users_guide.pdf>, 2008.

Microsoft Corporation. Windows registry information for advanced users. <http://support.microsoft.com/kb/256986>, 2008.

National Institute of Standards and Technology (NIST). The CFReDS project. <http://www.cfreds.nist.gov/>.

Petroni Jr NL, Fraser T, Walters A, Arbaugh WA. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In: USENIX-SS'06: Proceedings of the 15th Conference on USENIX Security Symposium. Berkeley, CA, USA: USENIX Association; 2006. p. 20.

Russinovich ME, Solomon DA. Microsoft Windows internals, Fourth edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (pro-developer). Redmond, WA, USA: Microsoft Press, ISBN 0735619174; 2004.

Samba. Regfio library. <http://viewcvs.samba.org/cgi-bin/viewcvs.cgi/branches/SAMBA_4_0/source/lib/registry/>.

Schuster A. Searching for processes and threads in Microsoft Windows memory dumps. In: Proceedings of the sixth Annual Digital Forensic Research Workshop (DFRWS 2006). <http://www.dfrws.org/2006/proceedings/2-Schuster.pdf>, 2006.

Schuster A. PoolTools version 1.3.0. <http://computer.forensikblog.de/en/2007/11/pooltools_1_3_0.html>, 2007.

Stevens D. UserAssist. <http://blog.didierstevens.com/programs/userassist/>, 2006.

Walters A. FATKit: detecting malicious library injection and upping the "anti", Technical report. 4TΦ Research Laboratories; July 2006.

Walters A. The Volatility framework: volatile memory artifact extraction utility framework. <https://www.volatilesystems.com/default/volatility>, 2007.

Walters A, Petroni NL, Jr., Volatools: integrating volatile memory forensics into the digital investigation process. In: Black Hat DC; 2007.

Brendan Dolan-Gavitt received a BA in Computer Science and Mathematics from Wesleyan University in Middletown, CT in 2006. He will be joining the Ph.D. Computer Science program at the Georgia Institute of Technology the fall of 2008.